Funkcje w C++ - podsumowanie

```
#include <cstdio>

int add(int a, int b) {
    return a + b;
}

int main() {
    printf("%i\n", add(4, 5));*wyszczególnienie*
    return 0;
}
```

Powyżej mamy 2 funkcje: `main` – funkcja główna i `add` – dodająca do siebie 2 liczby całkowite. `add` przyjmuje 2 argumenty:

- `int a`
- `int b`

W `main` uruchamiamy tę funkcję i przekazujemy do niej wartości 4 i 5, które w ciele funkcji stają się zmiennymi `a` i `b`. Zwracana jest wartość typu `int`, o czym świadczy słowo kluczowe `int` przed `add`.

return;

```
void func() {
    std::cout << "Hello, world!";
}
```

Ta funkcja nie zwraca żadnej wartości (`void` - pustka).

Zwrócenie wartości oznacza, że funkcja kończy swoje działanie i wraca tam, skąd została wywołana, a w miejscu wywołania pojawia się zwrócona wartość. Może ona zostać np. zapisana pod zmienną, przekazana jako argument kolejnego wywołania funkcji lub wykorzystana w instrukcji warunkowej `if`.

```
#include <iostream>
int main() {
    std::cout << "Ten tekst zostanie wyświetlony\n";
    return 0; // tu funkcja kończy swoje działanie
    std::cout << "Ten kod nigdy nie zostanie wykonany\n";
}
```

zmienne lokalne

Każda funkcja ma swoje własne zmienne. Jedynym sposobem na uzyskanie dostępu do danych spoza funkcji jest przekazanie ich do funkcji przez argumenty.

```
#include <cstdio>
void func(int a) {
    printf("%i\n", a); // 3
    a = 4;
    printf("%i\n", a); // 4

}
int main() {
    int a = 3;
    func();
    printf("%i\n", a); // 3
    return 0;
}
```

```
3
4
3
```

`func` nie zmienił wartości zmiennej `a` w `main`, tylko swojej własnej zmiennej. Przy odpalaniu funkcji C++ **kopiuje wartości do odpowiednich zmiennych lokalnych**.

Po zakończeniu funkcji jej pamięć jest zwalniana, co **psuje wszelkie wskaźniki i referencje** na dotychczas istniejące obiekty!

zwracanie wielu wartości & tablica jako argument

Do funkcji można przekazać referencję, co pozwoli jej na modyfikowanie oryginalnej zmiennej. Jest to przydatne gdy funkcja ma zwrócić wiele wartości.

```
#include <cstdio>
void sub_and_div(float a, float b, float &sub, float &div) {
    sub = a - b;
    div = a / b;
}

int main() {
    float sub, div;
    sub_and_div(3, 4, sub, div);
```

```
    printf("%f %f", sub, div); // -1 0.75
}
```

Tutaj, `sub_and_div` zapisuje wartości do zmiennych `sub` i `div`, ale są to referencje, przez które faktycznie zapisuje do oryginalnie przekazanych do funkcji zmiennych (zdeklarowanych w `main`).

Przekazując wskaźnik, również **umożliwiamy funkcji modyfikowanie wartości na którą wskazuje**, dlatego modyfikując elementy tablicy modyfikujemy oryginalne elementy. Jest to w zasadzie jedyny sposób zwrócenia tablicy, ponieważ **C++ nie pozwala zwrócić tablicy przez *return***.

```
#include <iostream>
// funkcja pomocnicza wyświetlająca tablicę
void print_arr(int arr[], int size) {
    for (int i=0; i<size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}

void func(int arr[], int size) {
    print_arr(arr, size); // 1 2 3 4 5
    for (int i=0; i<size; i++)
        arr[i] = 0;
    print_arr(arr, size); // 0 0 0 0 0
}

int main() {
    int arr[] = {1, 2, 3, 4, 5}, size = sizeof arr/sizeof arr[0];

    print_arr(arr, size); // 1 2 3 4 5
    func();
    print_arr(arr, size); // 0 0 0 0 0

    return 0;
}
```

Wewnątrz funkcji nie można określić liczby elementów tablicy przekazanej jako argument – wyrażenie `sizeof` zwróci jedynie długość adresu (zazwyczaj 8 bajtów). Rozmiar tablicy należy przekazać w oddzielnym argumencie.

argumenty domyślne

```
#include <cstdio>
void func(int a, int b, int c=1) {
    printf("%i %i %i", a, b, c);
```

```
}

int main() {
    func(0, 9); // 0 9 1 (odpowiednik func(0, 9, 1))
    func(3, 2, 7); // 3 2 7
    func(8); // BŁĄD (jaką wartość ma b?)

    return 0;
}
```

Nie można pomijać argumentów początkowych.

declarations & definitions

Deklaracja funkcji **daje znać kompilatorowi, że istnieje taka funkcja** – jest to pewnego rodzaju obietnica, że funkcja zostanie zdefiniowana później lub że została już skompilowana (np. jest częścią biblioteki). Na przykład:

```
int func1(char a[], char b[], int c=0);
void func2(float a[], int &b);
```

Kompilator czyta pliki od początku do końca – wyrzuci błąd gdy natrafi na wywołanie funkcji, która nie została jeszcze zdefiniowana lub zadeklarowana, nie patrząc czy nie ma tej definicji w dalszej części kodu. Deklaracje są w zasadzie niezbędne przy funkcjach, które wywołują siebie nawzajem:

```
void func2(int a);

void func1(int a) {
    if (a > 0)
        func2(--a) // func2 nie zostało jeszcze zdefiniowane, ale zostało
zadeklarowane
}

void func2(int a) {
    if (a > -1)
        func1(a-=2)
}
```

Jeżeli funkcja nigdy nie zostanie zdefiniowana, otrzymamy *błąd linkera* – na ostatnim etapie generowania pliku wykonywalnego.

inline

```
inline float pow(float base, unsigned int exponent) {
    float result = 1;
```

```
    for (int i=0; i<exponent; i++)
        result *= base;

    return result;
}
```

Różnice pomiędzy zwykłą funkcją a funkcją inline:

- kompilator decyduje, czy i w jaki sposób optymalizować daną funkcję,
- ciało funkcji musi być znane w miejscu wywołania (nie wystarczy deklaracja),
- celem jest wywołanie funkcji tak szybko, jak to możliwe.

zmienne statyczne

Zmienne statyczne, to zmienne wewnątrz funkcji, które zachowują swój stan pomiędzy kolejnymi wywołaniami funkcji.

```
int counter() {
    static int i = 0; // ustawiane na 0 tylko przy pierwszym wywołaniu
    return ++i; // zwracany jest numer wywołania liczony od 1
}
```

przeciążanie nazwy funkcji

Może istnieć kilka funkcji o tej samej nazwie, różniących się między sobą typami i liczbą argumentów. Kompilator rozróżnia odpowiednie funkcje przy wywołaniach po argumentach.

```
bool is_float(float a) {
    return true;
}

bool is_float(int a) {
    return false;
}
```

Dla zmiennych typu `float` zostanie odpalona funkcja 1, a dla zmiennych typu `int` funkcja 2. Obie funkcje noszą nazwę `is_float`.

main(argc, argv)

Jak wiemy, `main` jest naszym punktem wejścia, funkcją uruchamianą przez system operacyjny w momencie uruchamiania programu. Przekazuje on do `main` 2 argumenty, które możemy pobrać:

```
#include <cstdio>

int main(int argc, char* argv[]) {
    for (int i=0; i<argc; i++) {
        printf("%i. %s\n", i, argv[i]);
    }
}
```

Pierwszy z nich, to liczba argumentów wiersza poleceń, a drugi to tablica wskaźników na ciągi znaków (C string, null terminated) będących kolejnymi argumentami począwszy od nazwy pliku programu. To pozwala nam na przekazanie do programu dodatkowych parametrów w inny sposób niż poprzez stdin.

```
ls -l /etc
```

```
0. ls
1. -l
2. /etc
```

Brzmi znajomo?